

An Overview of Local Search Software Tools

Andrea Schaerf and Luca Di Gaspero

DIEGM - University of Udine

Introduction: Libraries, Frameworks, and Languages

Software Tools for Local Search

General Concepts

Overview of Existing Tools

- EASYLOCAL++
- ParadisEO
- Comet

Comparison between Tools

Introduction: Libraries, Frameworks, and Languages

Software Tools for Local Search

General Concepts

Overview of Existing Tools

- EASYLOCAL++

- ParadisEO

- Comet

Comparison between Tools

Introduction: Libraries, Frameworks, and Languages

Software Tools for Local Search

General Concepts

Overview of Existing Tools

- EASYLOCAL++
- ParadisEO
- Comet

Comparison between Tools

Introduction: Libraries, Frameworks, and Languages

Software Tools for Local Search

General Concepts

Overview of Existing Tools

- EASYLOCAL++
- ParadisEO
- Comet

Comparison between Tools

Introduction: Libraries, Frameworks, and Languages

Software Tools for Local Search

General Concepts

Overview of Existing Tools

- EASYLOCAL++

- ParadisEO

- Comet

Comparison between Tools

Introduction: Libraries, Frameworks, and Languages

Software Tools for Local Search

General Concepts

Overview of Existing Tools

- EASYLOCAL++
- ParadisEO
- Comet

Comparison between Tools

Introduction: Libraries, Frameworks, and Languages

Software Tools for Local Search

General Concepts

Overview of Existing Tools

- EASYLOCAL++
- ParadisEO
- Comet

Comparison between Tools

Introduction: Libraries, Frameworks, and Languages

Software Tools for Local Search

General Concepts

Overview of Existing Tools

- EASYLOCAL++
- ParadisEO
- Comet

Comparison between Tools

Introduction: Libraries, Frameworks, and Languages

Software Tools for Local Search

General Concepts

Overview of Existing Tools

- EASYLOCAL++
- ParadisEO
- Comet

Comparison between Tools

Software Libraries (Toolkits)

- ▶ Collections of subprograms
- ▶ Contain code that provides services to user's programs
- ▶ The user's code invokes the library (*direct control mechanism*)
- ▶ **Code** reuse

Software Libraries (Toolkits)

- ▶ Collections of subprograms
- ▶ Contain code that provides services to user's programs
- ▶ The user's code invokes the library (*direct control mechanism*)
- ▶ **Code** reuse

Software Libraries (Toolkits)

- ▶ Collections of subprograms
- ▶ Contain code that provides services to user's programs
- ▶ The user's code invokes the library (*direct control mechanism*)
- ▶ Code reuse

Software Libraries (Toolkits)

- ▶ Collections of subprograms
- ▶ Contain code that provides services to user's programs
- ▶ The user's code invokes the library (*direct control mechanism*)
- ▶ **Code** reuse

Software (Object-Oriented) Frameworks

- ▶ A set of abstract classes and their relations
- ▶ Rely on the *inverse control mechanism*
 - “**Hollywood Principle**: **Don't call us, we'll call you**”
the framework code calls the user-defined one
- ▶ The user defines the concrete subclasses and provides the requested methods:
 - the user defines the concrete classes that implement the abstract methods of the software system
 - the user defines the methods that implement the abstract methods of the user-defined classes
 - the user defines the methods that implement the abstract methods of the user-defined classes
 - the user defines the methods that implement the abstract methods of the user-defined classes
 - the user defines the methods that implement the abstract methods of the user-defined classes
- ▶ **Design reuse**

Software (Object-Oriented) Frameworks

- ▶ A set of abstract classes and their relations
- ▶ Rely on the *inverse control mechanism*
 - ▶ “**Hollywood Principle**: **Don’t call us, we’ll call you**”
the framework code calls the user-defined one
- ▶ The user defines the concrete subclasses and provides the requested methods:
 - ▶ *Frozen spots* (final methods): define the fixed overall architecture of the software system
 - ▶ *Abstract methods*: define the abstract methods that the user must implement
 - ▶ *Abstract classes*: define the abstract classes that the user must inherit
 - ▶ *Concrete classes*: define the concrete classes that the user must inherit
- ▶ **Design reuse**

Software (Object-Oriented) Frameworks

- ▶ A set of abstract classes and their relations
- ▶ Rely on the *inverse control mechanism*
 - ▶ “**Hollywood Principle**: **Don’t call us, we’ll call you**”
the framework code calls the user-defined one
- ▶ The user defines the concrete subclasses and provides the requested methods:
 - ▶ *Frozen spots* (final methods): define the fixed overall architecture of the software system
 - ▶ *Hot spots* (pure virtual methods): represent those parts where the user add her/his own code
 - ▶ *Cold spots* (virtual methods): represent those parts that the user may redefine
- ▶ **Design reuse**

Software (Object-Oriented) Frameworks

- ▶ A set of abstract classes and their relations
- ▶ Rely on the *inverse control mechanism*
 - ▶ “**Hollywood Principle**: **Don’t call us, we’ll call you**”
the framework code calls the user-defined one
- ▶ The user defines the concrete subclasses and provides the requested methods:
 - ▶ *Frozen spots* (final methods): define the fixed overall architecture of the software system
 - ▶ *Hot spots* (pure virtual methods): represent those parts where the user add her/his own code
 - ▶ *Cold spots* (virtual methods): represent those parts that the user may redefine
- ▶ Design reuse

Software (Object-Oriented) Frameworks

- ▶ A set of abstract classes and their relations
- ▶ Rely on the *inverse control mechanism*
 - ▶ “**Hollywood Principle**: **Don’t call us, we’ll call you**”
the framework code calls the user-defined one
- ▶ The user defines the concrete subclasses and provides the requested methods:
 - ▶ *Frozen spots* (final methods): define the fixed overall architecture of the software system
 - ▶ *Hot spots* (pure virtual methods): represent those parts where the user add her/his own code
 - ▶ *Cold spots* (virtual methods): represent those parts that the user may redefine
- ▶ **Design reuse**

Software (Object-Oriented) Frameworks

- ▶ A set of abstract classes and their relations
- ▶ Rely on the *inverse control mechanism*
 - “**Hollywood Principle**: **Don't call us, we'll call you**”
the framework code calls the user-defined one
- ▶ The user defines the concrete subclasses and provides the requested methods:
 - ▶ *Frozen spots* (final methods): define the fixed overall architecture of the software system
 - ▶ *Hot spots* (pure virtual methods): represent those parts where the user add her/his own code
 - ▶ *Cold spots* (virtual methods): represent those parts that the user may redefine
- ▶ **Design reuse**

Software (Object-Oriented) Frameworks

- ▶ A set of abstract classes and their relations
- ▶ Rely on the *inverse control mechanism*
 - “**Hollywood Principle**: **Don't call us, we'll call you**”
the framework code calls the user-defined one
- ▶ The user defines the concrete subclasses and provides the requested methods:
 - ▶ *Frozen spots* (final methods): define the fixed overall architecture of the software system
 - ▶ *Hot spots* (pure virtual methods): represent those parts where the user add her/his own code
 - ▶ *Cold spots* (virtual methods): represent those parts that the user may redefine
- ▶ **Design** reuse

Programming/Modeling Languages

- ▶ Languages with an *ad hoc* syntax and semantics
- ▶ Provide constructs for the problem modeling and/or the solution strategies
- ▶ Need a compiler/interpreter
- ▶ Difficult to embed in a host application
- ▶ **Knowledge** reuse

Programming/Modeling Languages

- ▶ Languages with an *ad hoc* syntax and semantics
- ▶ Provide constructs for the problem modeling and/or the solution strategies
- ▶ Need a compiler/interpreter
- ▶ Difficult to embed in a host application
- ▶ **Knowledge** reuse

Programming/Modeling Languages

- ▶ Languages with an *ad hoc* syntax and semantics
- ▶ Provide constructs for the problem modeling and/or the solution strategies
- ▶ Need a compiler/interpreter
- ▶ Difficult to embed in a host application
- ▶ **Knowledge** reuse

Programming/Modeling Languages

- ▶ Languages with an *ad hoc* syntax and semantics
- ▶ Provide constructs for the problem modeling and/or the solution strategies
- ▶ Need a compiler/interpreter
- ▶ Difficult to embed in a host application
- ▶ **Knowledge** reuse

Programming/Modeling Languages

- ▶ Languages with an *ad hoc* syntax and semantics
- ▶ Provide constructs for the problem modeling and/or the solution strategies
- ▶ Need a compiler/interpreter
- ▶ Difficult to embed in a host application
- ▶ **Knowledge** reuse

Introduction: Libraries, Frameworks, and Languages

Software Tools for Local Search

General Concepts

Overview of Existing Tools

- EASYLOCAL++
- ParadisEO
- Comet

Comparison between Tools

Software Tools for Local Search: Introduction

- ▶ For many optimization paradigms (e.g., ILP, CP, CLP, ...) there is a widely-accepted software tool (ILOG CPLEX, ILOG Solver, ECLiPSe, ...)
- ▶ Such a tool does not exist for Local Search:

- ▶ The apparent simplicity of Local Search induces the user to build the application from scratch

Software Tools for Local Search: Introduction

- ▶ For many optimization paradigms (e.g., ILP, CP, CLP, ...) there is a widely-accepted software tool (ILOG CPLEX, ILOG Solver, ECLiPSe, ...)
- ▶ Such a tool does not exist for Local Search:
 - ▶ The apparent simplicity of Local Search induces the user to build the application from scratch
 - ▶ The rapid evolution of the techniques makes the development of general tools harder
 - ▶ The lack of a unified view of Local Search makes it difficult to use other researchers' code (to this regard, the book (Hoos and Stützle, 2005) is very welcome)

Software Tools for Local Search: Introduction

- ▶ For many optimization paradigms (e.g., ILP, CP, CLP, ...) there is a widely-accepted software tool (ILOG CPLEX, ILOG Solver, ECLiPSe, ...)
- ▶ Such a tool does not exist for Local Search:
 - ▶ The apparent simplicity of Local Search induces the user to build the application from scratch
 - ▶ The rapid evolution of the techniques makes the development of general tools harder
 - ▶ The lack of a unified view of Local Search makes it difficult to use other researchers' code (to this regard, the book (Hoos and Stützle, 2005) is very welcome)

Software Tools for Local Search: Introduction

- ▶ For many optimization paradigms (e.g., ILP, CP, CLP, ...) there is a widely-accepted software tool (ILOG CPLEX, ILOG Solver, ECLiPSe, ...)
- ▶ Such a tool does not exist for Local Search:
 - ▶ The apparent simplicity of Local Search induces the user to build the application from scratch
 - ▶ The rapid evolution of the techniques makes the development of general tools harder
 - ▶ The lack of a unified view of Local Search makes it difficult to use other researchers' code (to this regard, the book (Hoos and Stützle, 2005) is very welcome)

Software Tools for Local Search: Introduction

- ▶ For many optimization paradigms (e.g., ILP, CP, CLP, ...) there is a widely-accepted software tool (ILOG CPLEX, ILOG Solver, ECLIPSe, ...)
- ▶ Such a tool does not exist for Local Search:
 - ▶ The apparent simplicity of Local Search induces the user to build the application from scratch
 - ▶ The rapid evolution of the techniques makes the development of general tools harder
 - ▶ The lack of a unified view of Local Search makes it difficult to use other researchers' code (to this regard, the book [\(Hoos and Stützle, 2005\)](#) is very welcome)

Software tools for Local Search Meta-heuristics

Tool	Reference	Language	Type	Paradigm
ILOG	(Shaw et al., 2002)	C++, Java, .NET	Tlkt	LS
GAlib	(Wall, 1996)	C++	Tlkt	GA
GAUL	(Adcock, 2005)	C	Tlkt	GA
Localizer++	(Michel and Van Hentenryck, 2000)	C++	Tlkt	Gen
HOTFRAME	(Fink and Voß, 2002)	C++	Frwk	LS
EASYLOCAL++	(Di Gaspero and Schaerf, 2003)	C++, Java	Frwk	LS
HSF	(Dorne and Voudouris, 2004)	Java	Frwk	LS, GA
ParadisEO	(Cahon et al., 2004)	C++	Frwk	EA, LS
OpenTS	(Harder et al., 2004)	Java	Frwk	TS
MDF	(Lau et al., 2007)	C++	Frwk	LS
TMF	(Watson, 2007)	C++	Frwk	LS
SALSA	(Laburthe and Caseau, 2002)	—	Lan	LS
Comet	(Van Hentenryck and Michel, 2005)	—	Lan	Gen

The state of the art in 2002: (Voß and Woodruff, 2002)

Software tools for Local Search Meta-heuristics

Tool	Reference	Language	Type	Paradigm
ILOG	(Shaw et al., 2002)	C++, Java, .NET	Tlkt	LS
GAlib	(Wall, 1996)	C++	Tlkt	GA
GAUL	(Adcock, 2005)	C	Tlkt	GA
Localizer++	(Michel and Van Hentenryck, 2000)	C++	Tlkt	Gen
HOTFRAME	(Fink and Voß, 2002)	C++	Frwk	LS
EASYLOCAL++	(Di Gaspero and Schaerf, 2003)	C++, Java	Frwk	LS
HSF	(Dorne and Voudouris, 2004)	Java	Frwk	LS, GA
ParadisEO	(Cahon et al., 2004)	C++	Frwk	EA, LS
OpenTS	(Harder et al., 2004)	Java	Frwk	TS
MDF	(Lau et al., 2007)	C++	Frwk	LS
TMF	(Watson, 2007)	C++	Frwk	LS
SALSA	(Laburthe and Caseau, 2002)	—	Lan	LS
Comet	(Van Hentenryck and Michel, 2005)	—	Lan	Gen

The state of the art in 2002: (Voß and Woodruff, 2002)

What's EASYLOCAL++

A C++ Object-Oriented framework for Local Search:

- ▶ Fully glass-box
- ▶ Decouples control-logic from problem representation (by means of generic programming)
- ▶ Manages problem-specific features by template instantiation and class derivation
- ▶ Based on Design Patterns (Decorator, Strategy and Template) and the Hollywood principle

Motivations

- ▶ Reuse & composition of code
- ▶ Conceptual clarity
- ▶ Prototyping and experimentation
- ▶ Design of novel techniques

What's EASYLOCAL++

A C++ Object-Oriented framework for Local Search:

- ▶ Fully glass-box
- ▶ Decouples control-logic from problem representation (by means of generic programming)
- ▶ Manages problem-specific features by template instantiation and class derivation
- ▶ Based on Design Patterns (Decorator, Strategy and Template) and the Hollywood principle

Motivations

- ▶ Reuse & composition of code
- ▶ Conceptual clarity
- ▶ Prototyping and experimentation
- ▶ Design of novel techniques

What's EASYLOCAL++

A C++ Object-Oriented framework for Local Search:

- ▶ Fully glass-box
- ▶ Decouples control-logic from problem representation (by means of generic programming)
- ▶ Manages problem-specific features by template instantiation and class derivation
- ▶ Based on Design Patterns (Decorator, Strategy and Template) and the Hollywood principle

Motivations

- ▶ Reuse & composition of code
- ▶ Conceptual clarity
- ▶ Prototyping and experimentation
- ▶ Design of novel techniques

What's EASYLOCAL++

A C++ Object-Oriented framework for Local Search:

- ▶ Fully glass-box
- ▶ Decouples control-logic from problem representation (by means of generic programming)
- ▶ Manages problem-specific features by template instantiation and class derivation
- ▶ Based on Design Patterns (Decorator, Strategy and Template) and the Hollywood principle

Motivations

- ▶ Reuse & composition of code
- ▶ Conceptual clarity
- ▶ Prototyping and experimentation
- ▶ Design of novel techniques

What's EASYLOCAL++

A C++ Object-Oriented framework for Local Search:

- ▶ Fully glass-box
- ▶ Decouples control-logic from problem representation (by means of generic programming)
- ▶ Manages problem-specific features by template instantiation and class derivation
- ▶ Based on Design Patterns (Decorator, Strategy and Template) and the Hollywood principle

Motivations

- ▶ Reuse & composition of code
- ▶ Conceptual clarity
- ▶ Prototyping and experimentation
- ▶ Design of novel techniques

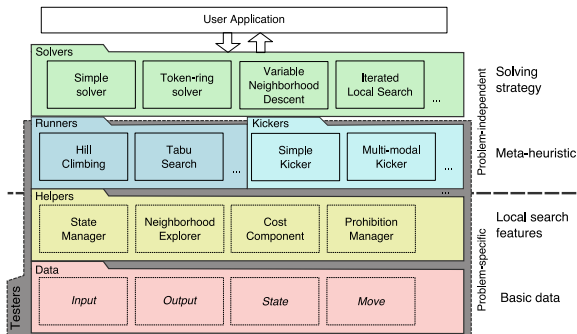
EASYLOCAL++ other features

- ▶ Developed in standard C++
- ▶ Small-sized (about 10,000 lines of code)
- ▶ Balanced use of Object-Oriented features
 - ▶ Template instantiation
 - ▶ Class derivation & virtual methods
- ▶ Efficient implementation (5–10% performance loss w.r.t. direct implementation)

Additional components/utilities of EASYLOCAL++

- ▶ Command-line parser for parameters
- ▶ Algorithm debugging and analysis support
 - ▶ Testers: basic user interface for test and debug
 - ▶ EASYANALYZER: a framework for behavior analyses
(Di Gaspero et al., 2007)
- ▶ Code generation
 - ▶ EASYSYN++: automatic code synthesis of EASYLOCAL++
hot spots (Di Gaspero and Schaerf, 2007)
- ▶ Foundation classes: Permutation, Partition, ...

EASYLOCAL++ architecture



Main *hot spots*:

- Data classes (template instantiation)
- Helpers (class derivation)

An example of EASYLOCAL++ abstract code

```
procedure LocalSearch( $S, N, F$ )  
begin
```

```
   $s_0 := \text{InitialSolution}(); i := 0;$   
  while ( $\neg \text{StopSearch}(s_i, i)$ ) do
```

```
     $m := \text{SelectMove}(s_i, F, N);$ 
```

```
    if ( $\text{AcceptableMove}($   
       $m, s_i, F)$ ) then
```

```
       $s_{i+1} := s_i \oplus m$ 
```

```
    end if;
```

```
     $i := i + 1$ 
```

```
  end while
```

```
end procedure
```

```
template <class Input, class State, class Move>
```

```
void MoveRunner<Input,State,Move>::Go()
```

```
{
```

```
  InitializeRun();
```

```
  while ( $\neg \text{StopCriterion}()$ ) {
```

```
    SelectMove();
```

```
    if ( $\text{AcceptableMove}()$ )
```

```
      MakeMove();
```

```
      UpdateIterationCounter();
```

```
  }
```

```
}
```

Instantiating the abstract algorithm

► Hill Climbing

```
template <class Input, class State, class Move>
bool HillClimbing<Input,State,Move>::AcceptableMove()
{ return (current_move_cost <= 0); }
```

► Simulated Annealing

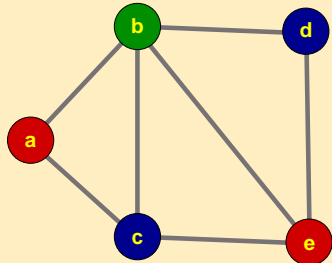
```
template <class Input, class State, class Move>
bool SimulatedAnnealing<Input,State,Move>::AcceptableMove()
{ return (current_move_cost <= 0) ||
    (Random::Float(0.0, 1.0) < exp(-current_move_cost/temperature)); }
```

► Tabu Search

```
template <class Input, class State, class Move>
bool TabuSearch<Input,State,Move>::AcceptableMove()
{ return !tabu_list.member(current_state, current_move) || tabu_list.
    Aspiration(current_state, best_state, current_move,
    current_move_cost); }
```

An example on the k -GRAPHCOLORING problem

Given an undirected graph $G = (V, E)$, and the set $\{0, \dots, k-1\}$ of color values, find an assignment of color to vertices such that adjacent vertices are assigned different colors.



Variables: $c_v, v \in V$

Domains: $\mathbb{D}_v = \{0, \dots, k-1\}$

Constraints: $\forall (u, v) \in E \quad c_u \neq c_v$

EASYLOCAL++ development flow

1. Define the data classes: Input, Output [▶ Go to](#)
2. Define the search space representation and the move(s) classes: State and Move
3. Define the strategy for constructing a state:
`StateManager::InitialState()`
4. Define the state evaluation class:
`CostComponent::ComputeCost()`
5. Define the neighborhood exploration strategies:
`NeighborhoodExplorer::RandomMove(), NextMove(),
MakeMove()`
6. Define the incremental state evaluation:
`DeltaCostComponent::ComputeDeltaCost()`
7. Define other suitable helpers: `TabuList::Inverse()`
8. Instantiate Runners and Solvers in a main program driver

EASYLOCAL++ development flow

1. Define the data classes: Input, Output
2. Define the search space representation and the move(s) classes: State and Move [▶ Go to](#)
3. Define the strategy for constructing a state:
`StateManager::InitialState()`
4. Define the state evaluation class:
`CostComponent::ComputeCost()`
5. Define the neighborhood exploration strategies:
`NeighborhoodExplorer::RandomMove(), NextMove(),
MakeMove()`
6. Define the incremental state evaluation:
`DeltaCostComponent::ComputeDeltaCost()`
7. Define other suitable helpers: `TabuList::Inverse()`
8. Instantiate Runners and Solvers in a main program driver

EASYLOCAL++ development flow

1. Define the data classes: `Input`, `Output`
2. Define the search space representation and the move(s) classes: `State` and `Move`

3. Define the strategy for constructing a state:

`StateManager::InitialState()` [▶ Go to](#)

4. Define the state evaluation class:

`CostComponent::ComputeCost()`

5. Define the neighborhood exploration strategies:

`NeighborhoodExplorer::RandomMove()`, `NextMove()`,
`MakeMove()`

6. Define the incremental state evaluation:

`DeltaCostComponent::ComputeDeltaCost()`

7. Define other suitable helpers: `TabuList::Inverse()`

8. Instantiate Runners and Solvers in a main program driver

EASYLOCAL++ development flow

1. Define the data classes: `Input`, `Output`
2. Define the search space representation and the move(s) classes: `State` and `Move`

3. Define the strategy for constructing a state:

`StateManager::InitialState()`

4. Define the state evaluation class:

`CostComponent::ComputeCost()` [▶ Go to](#)

5. Define the neighborhood exploration strategies:

`NeighborhoodExplorer::RandomMove()`, `NextMove()`,
`MakeMove()`

6. Define the incremental state evaluation:

`DeltaCostComponent::ComputeDeltaCost()`

7. Define other suitable helpers: `TabuList::Inverse()`

8. Instantiate Runners and Solvers in a main program driver

EASYLOCAL++ development flow

1. Define the data classes: `Input`, `Output`
2. Define the search space representation and the move(s) classes: `State` and `Move`
3. Define the strategy for constructing a state:
`StateManager::InitialState()`
4. Define the state evaluation class:
`CostComponent::ComputeCost()`
5. Define the neighborhood exploration strategies:
`NeighborhoodExplorer::RandomMove()`, `NextMove()`,
`MakeMove()` [▶ Go to](#)
6. Define the incremental state evaluation:
`DeltaCostComponent::ComputeDeltaCost()`
7. Define other suitable helpers: `TabuList::Inverse()`
8. Instantiate Runners and Solvers in a main program driver

EASYLOCAL++ development flow

1. Define the data classes: `Input`, `Output`
2. Define the search space representation and the move(s) classes: `State` and `Move`
3. Define the strategy for constructing a state:
`StateManager::InitialState()`
4. Define the state evaluation class:
`CostComponent::ComputeCost()`
5. Define the neighborhood exploration strategies:
`NeighborhoodExplorer::RandomMove()`, `NextMove()`,
`MakeMove()`
6. Define the incremental state evaluation:
`DeltaCostComponent::ComputeDeltaCost()` [Go to](#)
7. Define other suitable helpers: `TabuList::Inverse()`
8. Instantiate Runners and Solvers in a main program driver

EASYLOCAL++ development flow

1. Define the data classes: `Input`, `Output`
2. Define the search space representation and the move(s) classes: `State` and `Move`
3. Define the strategy for constructing a state:
`StateManager::InitialState()`
4. Define the state evaluation class:
`CostComponent::ComputeCost()`
5. Define the neighborhood exploration strategies:
`NeighborhoodExplorer::RandomMove()`, `NextMove()`,
`MakeMove()`
6. Define the incremental state evaluation:
`DeltaCostComponent::ComputeDeltaCost()`
7. Define other suitable helpers: `TabuList::Inverse()`

► Go to

8. Instantiate Runners and Solvers in a main program driver

EASYLOCAL++ development flow

1. Define the data classes: `Input`, `Output`
2. Define the search space representation and the move(s) classes: `State` and `Move`
3. Define the strategy for constructing a state:
`StateManager::InitialState()`
4. Define the state evaluation class:
`CostComponent::ComputeCost()`
5. Define the neighborhood exploration strategies:
`NeighborhoodExplorer::RandomMove()`, `NextMove()`,
`MakeMove()`
6. Define the incremental state evaluation:
`DeltaCostComponent::ComputeDeltaCost()`
7. Define other suitable helpers: `TabuList::Inverse()`
8. Instantiate Runners and Solvers in a main program driver [▶ Go to](#)

EASYLOCAL++ development flow

1. Define the data classes: `Input`, `Output`
2. Define the search space representation and the move(s) classes: `State` and `Move`
3. Define the strategy for constructing a state:
`StateManager::InitialState()`
4. Define the state evaluation class:
`CostComponent::ComputeCost()`
5. Define the neighborhood exploration strategies:
`NeighborhoodExplorer::RandomMove()`, `NextMove()`,
`MakeMove()`
6. Define the incremental state evaluation:
`DeltaCostComponent::ComputeDeltaCost()`
7. Define other suitable helpers: `TabuList::Inverse()`
8. Instantiate Runners and Solvers in a main program driver

Current/Future developments

- ▶ Neighborhood combination: union, composition
- ▶ High-level search strategies: Iterated Local Search, Variable Neighborhood Search, ...
- ▶ Automatic generation of the code (EASYSYN++)
- ▶ Hybridization support (with CP and/or LP)
- ▶ Support of sophisticated statistical analyses for algorithm comparison (EASYANALYZER)
- ▶ Learning techniques
- ▶ Foundation classes for different applicative domains (e.g., scheduling)

Current/Future developments

- ▶ Neighborhood combination: union, composition
- ▶ High-level search strategies: Iterated Local Search, Variable Neighborhood Search, ...
- ▶ Automatic generation of the code (EASYSYN++)
- ▶ Hybridization support (with CP and/or LP)
- ▶ Support of sophisticated statistical analyses for algorithm comparison (EASYANALYZER)
- ▶ Learning techniques
- ▶ Foundation classes for different applicative domains (e.g., scheduling)

Current/Future developments

- ▶ Neighborhood combination: union, composition
- ▶ High-level search strategies: Iterated Local Search, Variable Neighborhood Search, ...
- ▶ Automatic generation of the code (EASYSYN++)
- ▶ Hybridization support (with CP and/or LP)
- ▶ Support of sophisticated statistical analyses for algorithm comparison (EASYANALYZER)
- ▶ Learning techniques
- ▶ Foundation classes for different applicative domains (e.g., scheduling)

Current/Future developments

- ▶ Neighborhood combination: union, composition
- ▶ High-level search strategies: Iterated Local Search, Variable Neighborhood Search, ...
- ▶ Automatic generation of the code (EASYSYN++)
- ▶ Hybridization support (with CP and/or LP)
- ▶ Support of sophisticated statistical analyses for algorithm comparison (EASYANALYZER)
- ▶ Learning techniques
- ▶ Foundation classes for different applicative domains (e.g., scheduling)

Current/Future developments

- ▶ Neighborhood combination: union, composition
- ▶ High-level search strategies: Iterated Local Search, Variable Neighborhood Search, ...
- ▶ Automatic generation of the code (EASYSYN++)
- ▶ Hybridization support (with CP and/or LP)
- ▶ Support of sophisticated statistical analyses for algorithm comparison (EASYANALYZER)
- ▶ Learning techniques
- ▶ Foundation classes for different applicative domains (e.g., scheduling)

Current/Future developments

- ▶ Neighborhood combination: union, composition
- ▶ High-level search strategies: Iterated Local Search, Variable Neighborhood Search, ...
- ▶ Automatic generation of the code (EASYSYN++)
- ▶ Hybridization support (with CP and/or LP)
- ▶ Support of sophisticated statistical analyses for algorithm comparison (EASYANALYZER)
- ▶ Learning techniques
- ▶ Foundation classes for different applicative domains (e.g., scheduling)

Current/Future developments

- ▶ Neighborhood combination: union, composition
- ▶ High-level search strategies: Iterated Local Search, Variable Neighborhood Search, ...
- ▶ Automatic generation of the code (EASYSYN++)
- ▶ Hybridization support (with CP and/or LP)
- ▶ Support of sophisticated statistical analyses for algorithm comparison (EASYANALYZER)
- ▶ Learning techniques
- ▶ Foundation classes for different applicative domains (e.g., scheduling)

EASYLOCAL++ Availability

- ▶ Full version (totally revised): **Available.**
- ▶ Extra modules:
 - ▶ EASYANALYZER: : Available soon.
 - ▶ EASYST++: Available soon.

EASYLOCAL++ Availability

- ▶ Full version (totally revised): Available.
- ▶ Extra modules:
 - ▶ EASYANALYZER: Available soon.
 - ▶ EASYSYN++: Available not so soon.

EASYLOCAL++ Availability

- ▶ Full version (totally revised): Available.
- ▶ Extra modules:
 - ▶ EASYANALYZER: : Available soon.
 - ▶ EASYSYN++: Available not so soon.

EASYLOCAL++ Availability

- ▶ Full version (totally revised): Available.
- ▶ Extra modules:
 - ▶ EASYANALYZER: : Available soon.
 - ▶ EASYSYN++: Available not so soon.

What's ParadisEO?

- ▶ A C++ framework for Sequential and Parallel meta-heuristics
 - ▶ Genetic Algorithms, Particle Swarm optimization, Local Search
- ▶ Different packages:
 - ▶ Dynamic Multi-Objective
 - ▶ Moving Objects (local search)
 - ▶ Multi-objective Parallel Meta-Heuristics
 - ▶ Parallel Evolutionary Algorithms
- ▶ Shares many design ideas with EASYLOCAL++ (for the Moving Objects part)
- ▶ Includes a set of generic foundation components (e.g., chromosome arrays, etc.)
- ▶ Deeply based on functional objects (i.e., classes with a single responsibility)
- ▶ Parallelization is made transparent to the user

What's ParadisEO?

- ▶ A C++ framework for Sequential and Parallel meta-heuristics
 - ▶ Genetic Algorithms, Particle Swarm optimization, Local Search
- ▶ Different packages:
 - ▶ Evolutionary Objects
 - ▶ Moving Objects (Local search)
 - ▶ High-level Metaheuristics Objects
 - ▶ Parallel Evolutionary Objects
- ▶ Shares many design ideas with EASYLOCAL++ (for the Moving Objects part)
- ▶ Includes a set of generic foundation components (e.g., chromosome arrays, etc.)
- ▶ Deeply based on functional objects (i.e., classes with a single responsibility)
- ▶ Parallelization is made transparent to the user

What's ParadisEO?

- ▶ A C++ framework for Sequential and Parallel meta-heuristics
 - ▶ Genetic Algorithms, Particle Swarm optimization, Local Search
- ▶ Different packages:
 - ▶ Evolutionary Objects
 - ▶ Moving Objects (local search)
 - ▶ Multi-objective Evolutionary Objects
 - ▶ Parallel Evolutionary Objects
- ▶ Shares many design ideas with EASYLOCAL++ (for the Moving Objects part)
- ▶ Includes a set of generic foundation components (e.g., chromosome arrays, etc.)
- ▶ Deeply based on functional objects (i.e., classes with a single responsibility)
- ▶ Parallelization is made transparent to the user

What's ParadisEO?

- ▶ A C++ framework for Sequential and Parallel meta-heuristics
 - ▶ Genetic Algorithms, Particle Swarm optimization, Local Search
- ▶ Different packages:
 - ▶ Evolutionary Objects
 - ▶ Moving Objects (local search)
 - ▶ Multi-objective Evolutionary Objects
 - ▶ Parallel Evolutionary Objects
- ▶ Shares many design ideas with EASYLOCAL++ (for the Moving Objects part)
- ▶ Includes a set of generic foundation components (e.g., chromosome arrays, etc.)
- ▶ Deeply based on functional objects (i.e., classes with a single responsibility)
- ▶ Parallelization is made transparent to the user

What's ParadisEO?

- ▶ A C++ framework for Sequential and Parallel meta-heuristics
 - ▶ Genetic Algorithms, Particle Swarm optimization, Local Search
- ▶ Different packages:
 - ▶ Evolutionary Objects
 - ▶ Moving Objects (local search)
 - ▶ Multi-objective Evolutionary Objects
 - ▶ Parallel Evolutionary Objects
- ▶ Shares many design ideas with EASYLOCAL++ (for the Moving Objects part)
- ▶ Includes a set of generic foundation components (e.g., chromosome arrays, etc.)
- ▶ Deeply based on functional objects (i.e., classes with a single responsibility)
- ▶ Parallelization is made transparent to the user

What's ParadisEO?

- ▶ A C++ framework for Sequential and Parallel meta-heuristics
 - ▶ Genetic Algorithms, Particle Swarm optimization, Local Search
- ▶ Different packages:
 - ▶ Evolutionary Objects
 - ▶ Moving Objects (local search)
 - ▶ Multi-objective Evolutionary Objects
 - ▶ Parallel Evolutionary Objects
- ▶ Shares many design ideas with EASYLOCAL++ (for the Moving Objects part)
- ▶ Includes a set of generic foundation components (e.g., chromosome arrays, etc.)
- ▶ Deeply based on functional objects (i.e., classes with a single responsibility)
- ▶ Parallelization is made transparent to the user

What's ParadisEO?

- ▶ A C++ framework for Sequential and Parallel meta-heuristics
 - ▶ Genetic Algorithms, Particle Swarm optimization, Local Search
- ▶ Different packages:
 - ▶ Evolutionary Objects
 - ▶ Moving Objects (local search)
 - ▶ Multi-objective Evolutionary Objects
 - ▶ Parallel Evolutionary Objects
- ▶ Shares many design ideas with EASYLOCAL++ (for the Moving Objects part)
- ▶ Includes a set of generic foundation components (e.g., chromosome arrays, etc.)
- ▶ Deeply based on functional objects (i.e., classes with a single responsibility)
- ▶ Parallelization is made transparent to the user

What's ParadisEO?

- ▶ A C++ framework for Sequential and Parallel meta-heuristics
 - ▶ Genetic Algorithms, Particle Swarm optimization, Local Search
- ▶ Different packages:
 - ▶ Evolutionary Objects
 - ▶ Moving Objects (local search)
 - ▶ Multi-objective Evolutionary Objects
 - ▶ Parallel Evolutionary Objects
- ▶ Shares many design ideas with EASYLOCAL++ (for the Moving Objects part)
- ▶ Includes a set of generic foundation components (e.g., chromosome arrays, etc.)
- ▶ Deeply based on functional objects (i.e., classes with a single responsibility)
- ▶ Parallelization is made transparent to the user

What's ParadisEO?

- ▶ A C++ framework for Sequential and Parallel meta-heuristics
 - ▶ Genetic Algorithms, Particle Swarm optimization, Local Search
- ▶ Different packages:
 - ▶ Evolutionary Objects
 - ▶ Moving Objects (local search)
 - ▶ Multi-objective Evolutionary Objects
 - ▶ Parallel Evolutionary Objects
- ▶ Shares many design ideas with EASYLOCAL++ (for the Moving Objects part)
- ▶ Includes a set of generic foundation components (e.g., chromosome arrays, etc.)
- ▶ Deeply based on functional objects (i.e., classes with a single responsibility)
- ▶ Parallelization is made transparent to the user

What's ParadisEO?

- ▶ A C++ framework for Sequential and Parallel meta-heuristics
 - ▶ Genetic Algorithms, Particle Swarm optimization, Local Search
- ▶ Different packages:
 - ▶ Evolutionary Objects
 - ▶ Moving Objects (local search)
 - ▶ Multi-objective Evolutionary Objects
 - ▶ Parallel Evolutionary Objects
- ▶ Shares many design ideas with EASYLOCAL++ (for the Moving Objects part)
- ▶ Includes a set of generic foundation components (e.g., chromosome arrays, etc.)
- ▶ Deeply based on functional objects (i.e., classes with a single responsibility)
- ▶ Parallelization is made transparent to the user

What's ParadisEO?

- ▶ A C++ framework for Sequential and Parallel meta-heuristics
 - ▶ Genetic Algorithms, Particle Swarm optimization, Local Search
- ▶ Different packages:
 - ▶ Evolutionary Objects
 - ▶ Moving Objects (local search)
 - ▶ Multi-objective Evolutionary Objects
 - ▶ Parallel Evolutionary Objects
- ▶ Shares many design ideas with EASYLOCAL++ (for the Moving Objects part)
- ▶ Includes a set of generic foundation components (e.g., chromosome arrays, etc.)
- ▶ Deeply based on functional objects (i.e., classes with a single responsibility)
- ▶ Parallelization is made transparent to the user

The ParadisEO Tabu Search abstract code

```
template <class M>
bool moTS<M>::operator()(M::EOType& __sol) {
    if (__sol.invalid ())
        full_eval(__sol); // apply the full solution evaluation object
    M move;
    M::EOType best_sol = __sol, new_sol;
    cont.init(); // initialize the continuation condition object
    do {
        new_sol = __sol;
        try
            move_expl(__sol, new_sol); // apply the move loop explorer
        catch (EmptySelection& __ex)
            break;
        if (new_sol.fitness() > __sol.fitness())
            best_sol = new_sol;
        __sol = new_sol;
    } while (cont(__sol)); // check the continuation condition object
    __sol = best_sol;
    return true;
}
```

The Tabu Search move loop explorer

```
template <class M>
void moTSMoveLoopExpl<M>::operator()(const M::EOType& __old_sol, M::
    EOType& __new_sol) {
    M move, best_move;
    M::EOType::Fitness fit, best_move_fit;
    move_init(move, __old_sol); // restart the neighborhood exploration
    move_select.init(__old_sol.fitness()); // initialize the move selection criterion
    do {
        fit = incr_eval(move, __old_sol); // compute the neighbor fitness
        if (!tabu_list(move, __old_sol) || aspir_crit(move, fit))
            if (!move_select.update(move, fit)) // update the move selection object
                break;
    } while (next_move(move, __old_sol)); // go to the next neighbor
    move_select(best_move, best_move_fit);
    __new_sol.fitness(best_move_fit);
    best_move(__new_sol);
    tabu_list.update(); // update the tabu list (remove old tabu items)
    tabu_list.add(best_move, __new_sol); // add the current move to the tabu list
```

The Tabu Search move loop explorer

```
template <class M>
void moTSMoveLoopExpl<M>::operator()(const M::EOType& __old_sol, M::
    EOType& __new_sol) {
    M move, best_move;
    M::EOType::Fitness fit, best_move_fit;
    move_init(move, __old_sol); // restart the neighborhood exploration
    move_select.init(__old_sol.fitness()); // initialize the move selection criterion
    do {
        fit = incr_eval(move, __old_sol); // compute the neighbor fitness
        if (!tabu_list(move, __old_sol) || aspir_crit(move, fit))
            if (!move_select.update(move, fit)) // update the move selection object
                break;
    } while (next_move(move, __old_sol)); // go to the next neighbor
    move_select(best_move, best_move_fit);
    __new_sol.fitness(best_move_fit);
    best_move(__new_sol);
    tabu_list.update(); // update the tabu list (remove old tabu items)
    tabu_list.add(best_move, __new_sol); // add the current move to the tabu list
```

ParadisEO development flow

- ▶ Similar to the EASYLOCAL++ development flow.
- ▶ Main differences:
 - No explicit Input/Output objects, all is based on State and Move(s)
 - In the tutorial examples, the Input is a static class frequently employed by state
- ▶ Many class instantiations and derivations due to the functional object design choice
- ▶ One for each Local Search object
- ▶ An example of application code

ParadisEO development flow

- ▶ Similar to the EASYLOCAL++ development flow.
- ▶ Main differences:
 - ▶ No explicit Input/Output objects, all is based on State and Move(s)
 - ▶ In the tutorial examples, the Input is a static class implicitly employed by state
- ▶ Many class instantiations and derivations due to the functional object design choice
- ▶ One for each Local Search object
- ▶ An example of application code

ParadisEO development flow

- ▶ Similar to the EASYLOCAL++ development flow.
- ▶ Main differences:
 - ▶ No explicit Input/Output objects, all is based on State and Move(s)
 - ▶ In the tutorial examples, the Input is a static class implicitly employed by state
- ▶ Many class instantiations and derivations due to the functional object design choice
- ▶ An example of application code

ParadisEO development flow

- ▶ Similar to the EASYLOCAL++ development flow.
- ▶ Main differences:
 - ▶ No explicit Input/Output objects, all is based on State and Move(s)
 - ▶ In the tutorial examples, the Input is a static class implicitly employed by state
- ▶ Many class instantiations and derivations due to the functional object design choice
 - ▶ one for each Local Search aspect
- ▶ An example of application code

ParadisEO development flow

- ▶ Similar to the EASYLOCAL++ development flow.
- ▶ Main differences:
 - ▶ No explicit Input/Output objects, all is based on State and Move(s)
 - ▶ In the tutorial examples, the Input is a static class implicitly employed by state
- ▶ Many class instantiations and derivations due to the functional object design choice
 - ▶ one for each Local Search aspect
- ▶ An example of application code

ParadisEO development flow

- ▶ Similar to the EASYLOCAL++ development flow.
- ▶ Main differences:
 - ▶ No explicit Input/Output objects, all is based on State and Move(s)
 - ▶ In the tutorial examples, the Input is a static class implicitly employed by state
- ▶ Many class instantiations and derivations due to the functional object design choice
 - ▶ one for each Local Search aspect
- ▶ An example of application code

ParadisEO development flow

- ▶ Similar to the EASYLOCAL++ development flow.
- ▶ Main differences:
 - ▶ No explicit Input/Output objects, all is based on State and Move(s)
 - ▶ In the tutorial examples, the Input is a static class implicitly employed by state
- ▶ Many class instantiations and derivations due to the functional object design choice
 - ▶ one for each Local Search aspect
- ▶ An example of application code [▶ Go to](#)

ParadisEO development flow

- ▶ Similar to the EASYLOCAL++ development flow.
- ▶ Main differences:
 - ▶ No explicit Input/Output objects, all is based on State and Move(s)
 - ▶ In the tutorial examples, the Input is a static class implicitly employed by state
- ▶ Many class instantiations and derivations due to the functional object design choice
 - ▶ one for each Local Search aspect
- ▶ An example of application code

Languages for Combinatorial Optimisation Problems

Integer Programming: linear constraints expressed using algebraic and set notations

Constraint Programming: constraints expressed and combined at high level of abstraction

Constraint-Based Local Search uses constraints to describe and control local search.

Languages for Combinatorial Optimisation Problems

Integer Programming: linear constraints expressed using algebraic and set notations

Constraint Programming: constraints expressed and combined at high level of abstraction

Constraint-Based Local Search uses constraints to describe and control local search.

Languages for Combinatorial Optimisation Problems

Integer Programming: linear constraints expressed using algebraic and set notations

Constraint Programming: constraints expressed and combined at high level of abstraction

Constraint-Based Local Search uses constraints to describe and control local search.

Languages for Combinatorial Optimisation Problems

Integer Programming: linear constraints expressed using algebraic and set notations

Constraint Programming: constraints expressed and combined at high level of abstraction

Constraint-Based Local Search uses constraints to describe and control local search.

What's Comet?

- ▶ An Object-Oriented language with innovative modeling and control abstractions for Local Search (mainly declarative)
 - ▶ invariants, numerical, logical, and combinatorial constraints and objectives
 - ▶ randomized selectors, events, checkpoints, neighbors
- ▶ Support of scheduling abstractions
- ▶ Support of a wide range of scheduling algorithms
- ▶ Support of a wide range of search strategies
- ▶ Closed source, black box, binary available only for a number of systems

What's Comet?

- ▶ An Object-Oriented language with innovative modeling and control abstractions for Local Search (mainly declarative)
 - ▶ invariants, numerical, logical, and combinatorial constraints and objectives
 - ▶ randomized selectors, events, checkpoints, neighbors
- ▶ Support of scheduling abstractions
 - ▶ schedules, jobs, activities, precedence constraints, resources (cumulative/disjunctive), objective functions (makespan, tardiness)
- ▶ Closed source, black box, binary available only for a number of systems

What's Comet?

- ▶ An Object-Oriented language with innovative modeling and control abstractions for Local Search (mainly declarative)
 - ▶ invariants, numerical, logical, and combinatorial constraints and objectives
 - ▶ randomized selectors, events, checkpoints, neighbors
- ▶ Support of scheduling abstractions
 - ▶ schedules, jobs, activities, precedence constraints, resources (cumulative/disjunctive), objective functions (makespan, tardiness)
- ▶ Closed source, black box, binary available only for a number of systems

What's Comet?

- ▶ An Object-Oriented language with innovative modeling and control abstractions for Local Search (mainly declarative)
 - ▶ invariants, numerical, logical, and combinatorial constraints and objectives
 - ▶ randomized selectors, events, checkpoints, neighbors
- ▶ Support of scheduling abstractions
 - ▶ schedules, jobs, activities, precedence constraints, resources (cumulative/disjunctive), objective functions (makespan, tardiness)
- ▶ Closed source, black box, binary available only for a number of systems

What's Comet?

- ▶ An Object-Oriented language with innovative modeling and control abstractions for Local Search (mainly declarative)
 - ▶ invariants, numerical, logical, and combinatorial constraints and objectives
 - ▶ randomized selectors, events, checkpoints, neighbors
- ▶ Support of scheduling abstractions
 - ▶ schedules, jobs, activities, precedence constraints, resources (cumulative/disjunctive), objective functions (makespan, tardiness)
- ▶ Closed source, black box, binary available only for a number of systems

The architecture of Comet

Local Search = model + search

Modelling part: purely declarative, expresses the combinatorial structure of the application in terms of constraints and objective function

Search component: exploits the structure expressed in the model to guide the neighbourhood exploration towards high quality solutions

The architecture of Comet

$$\textit{Local Search} = \textit{model} + \textit{search}$$

Modelling part: purely declarative, expresses the combinatorial structure of the application in terms of constraints and objective function

Search component: exploits the structure expressed in the model to guide the neighbourhood exploration towards high quality solutions

The architecture of Comet

$$\textit{Local Search} = \textit{model} + \textit{search}$$

Modelling part: purely declarative, expresses the combinatorial structure of the application in terms of constraints and objective function

Search component: exploits the structure expressed in the model to guide the neighbourhood exploration towards high quality solutions

Benefits of Comet

- ▶ Easy combination of constraints and objective through arithmetic, logical and cardinality operators
- ▶ Separation of model and search → modularity and reuse
- ▶ Extensibility and flexibility
- ▶ Technology-independent

Benefits of Comet

- ▶ Easy combination of constraints and objective through arithmetic, logical and cardinality operators
- ▶ Separation of model and search → modularity and reuse
- ▶ Extensibility and flexibility
- ▶ Technology-independent

Benefits of Comet

- ▶ Easy combination of constraints and objective through arithmetic, logical and cardinality operators
- ▶ Separation of model and search → modularity and reuse
- ▶ Extensibility and flexibility
- ▶ Technology-independent

Benefits of Comet

- ▶ Easy combination of constraints and objective through arithmetic, logical and cardinality operators
- ▶ Separation of model and search → modularity and reuse
- ▶ Extensibility and flexibility
- ▶ Technology-independent

Comet development flow

- ▶ Define the problem model [▶ Go to](#)
- ▶ Define the Local Search model
- ▶ Code the Local Search meta-heuristic

Comet development flow

- ▶ Define the problem model
- ▶ Define the Local Search model ▶ Go to
- ▶ Code the Local Search meta-heuristic

Comet development flow

- ▶ Define the problem model
- ▶ Define the Local Search model
- ▶ Code the Local Search meta-heuristic [▶ Go to](#)

Introduction: Libraries, Frameworks, and Languages

Software Tools for Local Search

General Concepts

Overview of Existing Tools

- EASYLOCAL++
- ParadisEO
- Comet

Comparison between Tools

Tools comparison

	EASYLOCAL++	ParadisEO	Comet
Learnability	Must understand the framework control flow		Some features require a careful understanding of the semantics
Code reuse	Framework level and (partially) application level		Limited
Flexibility	Problem-specific features must be implemented in the “right place”, new algorithms must fit within the general architecture		Full flexibility, also on problem-specific features
Integrability	Full access to external libraries/software		No integration
Testing & Debugging	Dedicated components (Testers)	Dedicated components (Monitors)	GUI debugger
Parallel execution	Experimental	Full supported	Under testing
Statistical analysis	EASYANALYZER	(EASYANALYZER (future development))	—
Code generation and management	EASYSYN++ (under development)	—	—

Tools comparison

	EASYLOCAL++	ParadisEO	Comet
Learnability	Must understand the framework control flow		Some features require a careful understanding of the semantics
Code reuse	Framework level and (partially) application level		Limited
Flexibility	Problem-specific features must be implemented in the “right place”, new algorithms must fit within the general architecture		Full flexibility, also on problem-specific features
Integrability	Full access to external libraries/software		No integration
Testing & Debugging	Dedicated components (Testers)	Dedicated components (Monitors)	GUI debugger
Parallel execution	Experimental	Full supported	Under testing
Statistical analysis	EASYANALYZER	(EASYANALYZER (future development))	—
Code generation and management	EASYSYN++ (under development)	—	—

Tools comparison

	EASYLOCAL++	ParadisEO	Comet
Learnability	Must understand the framework control flow		Some features require a careful understanding of the semantics
Code reuse	Framework level and (partially) application level		Limited
Flexibility	Problem-specific features must be implemented in the “right place”, new algorithms must fit within the general architecture		Full flexibility, also on problem-specific features
Integrability	Full access to external libraries/software		No integration
Testing & Debugging	Dedicated components (Testers)	Dedicated components (Monitors)	GUI debugger
Parallel execution	Experimental	Full supported	Under testing
Statistical analysis	EASYANALYZER	(EASYANALYZER (future development))	—
Code generation and management	EASYSYN++ (under development)	—	—

Tools comparison

	EASYLOCAL++	ParadisEO	Comet
Learnability	Must understand the framework control flow		Some features require a careful understanding of the semantics
Code reuse	Framework level and (partially) application level		Limited
Flexibility	Problem-specific features must be implemented in the “right place”, new algorithms must fit within the general architecture		Full flexibility, also on problem-specific features
Integrability	Full access to external libraries/software		No integration
Testing & Debugging	Dedicated components (Testers)	Dedicated components (Monitors)	GUI debugger
Parallel execution	Experimental	Full supported	Under testing
Statistical analysis	EASYANALYZER	(EASYANALYZER (future development))	—
Code generation and management	EASYSYN++ (under development)	—	—

Tools comparison

	EASYLOCAL++	ParadisEO	Comet
Learnability	Must understand the framework control flow		Some features require a careful understanding of the semantics
Code reuse	Framework level and (partially) application level		Limited
Flexibility	Problem-specific features must be implemented in the “right place”, new algorithms must fit within the general architecture		Full flexibility, also on problem-specific features
Integrability	Full access to external libraries/software		No integration
Testing & Debugging	Dedicated components (Testers)	Dedicated components (Monitors)	GUI debugger
Parallel execution	Experimental	Full supported	Under testing
Statistical analysis	EASYANALYZER	(EASYANALYZER (future development))	—
Code generation and management	EASYSYN++ (under development)	—	—

Tools comparison

	EASYLOCAL++	ParadisEO	Comet
Learnability	Must understand the framework control flow		Some features require a careful understanding of the semantics
Code reuse	Framework level and (partially) application level		Limited
Flexibility	Problem-specific features must be implemented in the “right place”, new algorithms must fit within the general architecture		Full flexibility, also on problem-specific features
Integrability	Full access to external libraries/software		No integration
Testing & Debugging	Dedicated components (Testers)	Dedicated components (Monitors)	GUI debugger
Parallel execution	Experimental	Full supported	Under testing
Statistical analysis	EASYANALYZER	(EASYANALYZER (future development))	—
Code generation and management	EASYSYN++ (under development)	—	—

Tools comparison

	EASYLOCAL++	ParadisEO	Comet
Learnability	Must understand the framework control flow		Some features require a careful understanding of the semantics
Code reuse	Framework level and (partially) application level		Limited
Flexibility	Problem-specific features must be implemented in the “right place”, new algorithms must fit within the general architecture		Full flexibility, also on problem-specific features
Integrability	Full access to external libraries/software		No integration
Testing & Debugging	Dedicated components (Testers)	Dedicated components (Monitors)	GUI debugger
Parallel execution	Experimental	Full supported	Under testing
Statistical analysis	EASYANALYZER	(EASYANALYZER (future development))	—
Code generation and management	EASYSYN++ (under development)	—	—

Tools comparison

	EASYLOCAL++	ParadisEO	Comet
Learnability	Must understand the framework control flow		Some features require a careful understanding of the semantics
Code reuse	Framework level and (partially) application level		Limited
Flexibility	Problem-specific features must be implemented in the “right place”, new algorithms must fit within the general architecture		Full flexibility, also on problem-specific features
Integrability	Full access to external libraries/software		No integration
Testing & Debugging	Dedicated components (Testers)	Dedicated components (Monitors)	GUI debugger
Parallel execution	Experimental	Full supported	Under testing
Statistical analysis	EASYANALYZER	(EASYANALYZER (future development))	—
Code generation and management	EASYSYN++ (under development)	—	—

Tools comparison

	EASYLOCAL++	ParadisEO	Comet
Learnability	Must understand the framework control flow		Some features require a careful understanding of the semantics
Code reuse	Framework level and (partially) application level		Limited
Flexibility	Problem-specific features must be implemented in the “right place”, new algorithms must fit within the general architecture		Full flexibility, also on problem-specific features
Integrability	Full access to external libraries/software		No integration
Testing & Debugging	Dedicated components (Testers)	Dedicated components (Monitors)	GUI debugger
Parallel execution	Experimental	Full supported	Under testing
Statistical analysis	EASYANALYZER	(EASYANALYZER (future development))	—
Code generation and management	EASYSYN++ (under development)	—	—

Tools performances

Instance	Cols	EASYLOCAL++		ParadisEO		Comet	
		<u>Time</u>	<u>Cost</u>	<u>Time</u>	<u>Cost</u>	<u>Time</u>	<u>Cost</u>
DSJC125.1.col	5	1.60*	0.00*	1.68*	0.00*	1.65*	0.00*
DSJC125.5.col	17	45.14*	0.26*	44.12*	0.24*	43.99*	0.27*
DSJC125.9.col	44	7.08*	0.00*	7.52*	0.00*	7.27*	0.00*
DSJC250.1.col	8	25.72*	0.44*	26.37*	0.44*	25.77*	0.43*
DSJC250.5.col	28	337.75*	4.72*	358.67*	4.90*	352.95*	4.67*
DSJC250.9.col	72	363.21*	0.79*	345.75*	0.80*	346.88*	0.76*
DSJC500.1.col	12	350.93*	18.19*	353.99*	19.13*	340.88*	18.47*
DSJC500.5.col	48	2045.32*	25.62*	2060.16*	26.54*	1984.87*	25.40*
DSJC500.9.col	126	2875.15*	9.68*	3018.63*	9.87*	2974.45*	9.59*

* Pairwise Mann-Whitney tests could not reject the null hypothesis of the difference in the distributions ($\alpha = 0.05$)

	EASYLOCAL++	ParadisEO	Comet
Lines of code	570	540	110

Tools performances

Instance	Cols	EASYLOCAL++		ParadisEO		Comet	
		<u>Time</u>	<u>Cost</u>	<u>Time</u>	<u>Cost</u>	<u>Time</u>	<u>Cost</u>
DSJC125.1.col	5	1.60*	0.00*	1.68*	0.00*	1.65*	0.00*
DSJC125.5.col	17	45.14*	0.26*	44.12*	0.24*	43.99*	0.27*
DSJC125.9.col	44	7.08*	0.00*	7.52*	0.00*	7.27*	0.00*
DSJC250.1.col	8	25.72*	0.44*	26.37*	0.44*	25.77*	0.43*
DSJC250.5.col	28	337.75*	4.72*	358.67*	4.90*	352.95*	4.67*
DSJC250.9.col	72	363.21*	0.79*	345.75*	0.80*	346.88*	0.76*
DSJC500.1.col	12	350.93*	18.19*	353.99*	19.13*	340.88*	18.47*
DSJC500.5.col	48	2045.32*	25.62*	2060.16*	26.54*	1984.87*	25.40*
DSJC500.9.col	126	2875.15*	9.68*	3018.63*	9.87*	2974.45*	9.59*

* Pairwise Mann-Whitney tests could not reject the null hypothesis of the difference in the distributions ($\alpha = 0.05$)

	EASYLOCAL++	ParadisEO	Comet
Lines of code	570	540	110

Thanks for your attention

Input class

```
class Graph {  
    friend std::ostream& operator<<(ostream&, const Graph&);  
public:  
    typedef unsigned Color;  
    typedef unsigned Vertex;  
    // constructs a graph from a DIMACS file  
    Graph(const string& filename, Color ncols);  
    bool Edge(const Vertex& v, const Vertex& w) const;  
    unsigned Vertices() const;  
    unsigned Edges() const;  
    unsigned k;  
protected:  
    vector<vector<bool> > adjacency_matrix;  
    unsigned num_of_vertices;  
    unsigned num_of_edges;  
};
```

Output class

```
class Coloring {  
public:  
    Coloring(const Graph& g_in)  
        : g(g_in), coloring(g.Vertices()) {}  
    Color& operator()(unsigned i) { return coloring(i); }  
    Color operator()(unsigned i) const { return coloring(i); }  
    unsigned size() const { return coloring.size(); }  
private  
    const Graph& g;  
    vector<Color> coloring;  
};
```

◀ Go back

State

```
class ColorState : public Coloring {
    friend ostream& operator<<(ostream&, const ColorState&);
public:
    ColorState(const Graph& in);
    bool InConflicts(const Vertex& v) const
    { return conflicts(v) > 0; }
    int Conflicts(const Vertex& v) const
    { return conflicts(v); }
    unsigned ConflictsSize() const
    { return conflicts_size; }
    void AddToConflicts(const Vertex& v);
    void RemoveFromConflicts(const Vertex& v)
    void UpdateRedundantData();
protected:
    vector<unsigned> conflicts; // a map  $V \rightarrow \mathbb{N}$  with the number of conflicts
    unsigned conflicts_size; // number of non-zero entries of the previous map
};
```

Move

```
class Recolor {  
    friend ostream& operator<<(ostream&, const Recolor&);  
    friend istream& operator>>(istream&, Recolor&);  
    friend bool operator==(const Recolor&,const Recolor&);  
    friend bool operator!=(const Recolor&,const Recolor&);  
    friend bool operator<(const Recolor&,const Recolor&);  
public:  
    Vertex v; // the vertex  
    Color nc; // the new color assigned to the vertex  
    Color oc; // the old color  
};
```

◀ Go back

State manager

```
ColorStateManager::ColorStateManager(const Graph& in) : StateManager<
    Graph, ColorState>(in) {}
```

```
// initial state builder
```

```
void StateManager::InitialState(ColorState& coloring)
```

```
{
```

```
    // assign all nodes randomly
```

```
    for (Vertex v = 0; v < in.Vertices(); v++)
```

```
        coloring(v) = Random::Int(0, in.k - 1);
```

```
    // update redundant state data
```

```
    coloring.UpdateRedundantData();
```

```
}
```

◀ Go back

Cost Component

```
class ConflictSize : public CostComponent<Graph, ColorState, int> {  
public:  
    ConflictSize(const Graph& in) : CostComponent<Graph,ColorState>(in, 1,  
        false, "Conflict_Size") {}  
    int ComputeCost(const ColorState& st) const  
    { return st.ConflictSize(); }  
};
```

◀ Go back

Neighborhood Explorer

```
void RecolorExplorer::RandomMove(const ColorState& coloring, Recolor& rc) {  
    rc.v = Random::Int(0, in.Vertices() - 1);  
    while (!coloring.InConflicts(rc.v))  
        rc.v = (rc.v + 1) % in.Vertices();  
    rc.oc = coloring(rc.v);  
    do rc.nc = (Color)Random::Int(0, in.k - 1);  
    while (rc.nc == rc.oc);  
}
```

```
void RecolorExplorer::NextMove(const ColorState& coloring, Recolor& rc) {  
    rc.nc = (rc.nc + 1) % in.k;  
    // try the next conflicting node skipping the trivial move (i.e. nc = oc)  
    if (rc.nc == rc.oc) {  
        do rc.v = (rc.v + 1) % in.Vertices();  
        while (!coloring.InConflicts(rc.v));  
        rc.oc = coloring(rc.v);  
        rc.nc = (rc.oc + 1) % in.k;  
    }  
}
```

Delta Cost Component

```
int DeltaConflictSize::ComputeDeltaCost(const ColorState& coloring, const
    Recolor& rc) const {
    int delta_cost = 0, conflicts_v = coloring.Conflicts(rc.v);
    for (Vertex w = 0; w < in.Vertices(); w++)
        if (in.Edge(rc.v, w)) {
            if (coloring(w) == rc.nc) {
                if (coloring.Conflicts(w) == 0) delta_cost++;
                conflicts_v++;
            }
            if (coloring(w) == rc.oc) {
                if (coloring.Conflicts(w) == 1) delta_cost--;
                conflicts_v--;
            }
        }
    if (conflicts_v == 0) delta_cost--;
    return delta_cost;
}
```


Tabu List Manager

```
bool ProhibitedColorsManager::Inverse(const Recolor& rc1, const Recolor& rc2)
    const
{ return (rc1.v == rc2.v && rc1.nc == rc2.oc); }

// For the [Dorne & Hao, 1998] dynamic length  $\alpha \cdot |\text{conflicts}| + \text{rand}(\text{min}, \text{max})$ 
void ProhibitedColorsManager::InsertIntoList(const ColorState& coloring, const
    Recolor& rc) {
    unsigned tenure = (alpha * coloring.ConflictsSize()) + Random::Int(min_tenure
        , max_tenure);
    TabuListItem<ColorState, Recolor, int> li(rc, iter + tenure);
    tlist.push_front(li);
    UpdateIteration();
}
```

◀ Go back

Object instantiation

```
// Data and Helpers
Graph g(argFilename.getValue(), argColors.getValue());
ColorStateManager sm(g);
ConflictSize cc(g); sm.AddCostComponent(cc);
RecolorExplorer ne(g, sm);
DeltaConflictSize dcc(g, cc); ne.AddDeltaCostComponent(dcc);
ProhibitedColorsManager tlm(argAlpha.getValue(), argTabuList.getValue(0),
    argTabuList.getValue(1));

// Runners
HillClimbing<Graph, ColorState, Recolor> hc(g, sm, ne);
hc.SetMaxIdleIteration(argIdleIterations.getValue());
TabuSearch<Graph, ColorState, Recolor> ts(g, sm, ne, tlm);
ts.SetMaxIdleIteration(argIdleIterations.getValue());

// Solvers
SimpleLocalSearch<Graph, Coloring, ColorState> simple_solver(g, sm, om);

// Running
simple_solver.SetRunner(ts);
simple_solver.Solve();
Output out = simple_solver.GetOutput();
```

ParadisEO problem model

```

Recolor Recolor::operator!() const {
    swap(this->oc, this->nc);
    return *this;
}

void Recolor::operator()(Coloring& coloring) {
    if (nc == oc) return; // nothing to be done for the identical move
    coloring(v) = nc; // update the color
    for (Vertex w = 0; w < Graph::Vertices(); w++) // update the conflict list
        if (Graph::Edge(v, w)) {
            if (coloring(w) == nc) { // a new conflict has found
                coloring.add_to_conflicts(v);
                coloring.add_to_conflicts(w);
            }
            if (coloring(w) == oc) { // an old conflict has removed
                coloring.remove_from_conflicts(v);
                coloring.remove_from_conflicts(w);
            }
        }
}

```

Comet problem model

```
range Vertices = 1..n;
range Colors = 0..k-1;
bool adj(Vertices, Vertices) = false;
/* Reading the DIMACS file [omitted] */
UniformDistribution d(Colors);
LocalSolver mgr();
var{int} coloring(Vertices)(mgr, Colors) := d.get();
DisequationSystem S(coloring);
// Stating coloring[u] ≠ coloring[v] for (u, v) ∈ E
forall(u in Vertices, v in Vertices : v > u && adj(u, v))
    S.post(coloring(u), coloring(v));
S.close();

var{int} conflicts = S.violations();
var{int} nb_conflicts(v in Vertices) = S.violations(coloring(v));
var{set{int}} conflict_list(mgr) <- setof(v in Vertices) (nb_conflicts(v) > 0);
mgr.close();
```

Comet problem model

```
range Vertices = 1..n;
range Colors = 0..k-1;
bool adj(Vertices, Vertices) = false;
/* Reading the DIMACS file [omitted] */
UniformDistribution d(Colors);
LocalSolver mgr();
var{int} coloring(Vertices)(mgr, Colors) := d.get();
DisequationSystem S(coloring);
// Stating coloring[u] ≠ coloring[v] for (u, v) ∈ E
forall(u in Vertices, v in Vertices : v > u && adj(u, v))
    S.post(coloring(u), coloring(v));
S.close();

var{int} conflicts = S.violations();
var{int} nb_conflicts(v in Vertices) = S.violations(coloring(v));
var{set{int}} conflict_list(mgr) <- setof(v in Vertices) (nb_conflicts(v) > 0);
mgr.close();
```

Comet Local Search model

```

int max_iter = 10000;
int max_idle_iter = 300000;
int min_tenure = 1;
int max_tenure = 10;
int alpha = 2;
int best_value = conflicts;
Solution best_solution(mgr);
UniformDistribution r(min_tenure..max_tenure); // for the tabu length
Counter idle_iter(mgr) := 0;
Counter iter(mgr) := 0;
bool tabu(Vertices, Colors) = false;
// Use events and closures for managing the tabu list
forall (v in Vertices)
    whenever coloring(v)@changes(int oc, int nc) {
        int tenure = iter + r.get() + (alpha * card(conflict_list));
        tabu(v, oc) = true;
        when iter@reaches(tenure)() tabu(v, oc) = false;
    }

```

Comet Local Search model

```

int max_iter = 10000;
int max_idle_iter = 300000;
int min_tenure = 1;
int max_tenure = 10;
int alpha = 2;
int best_value = conflicts;
Solution best_solution(mgr);
UniformDistribution r(min_tenure..max_tenure); // for the tabu length
Counter idle_iter(mgr) := 0;

Counter iter(mgr) := 0;
bool tabu(Vertices, Colors) = false;
// Use events and closures for managing the tabu list
forall (v in Vertices)
  whenever coloring(v)@changes(int oc, int nc) {
    int tenure = iter + r.get() + (alpha * card(conflict_list));
    tabu(v, oc) = true;
    when iter@reaches(tenure)() tabu(v, oc) = false;
  }

```

Comet Local Search strategy

```
while (conflicts > 0 && idle_iter < max_idle_iter) {  
    int gap_to_best = best_value – conflicts;  
    selectMin(v in conflict_list, c in Colors, d = S.getAssignDelta(coloring(v), c) : !  
        tabu(v, c) || d < gap_to_best)(d)  
    coloring(v) := c;  
    if (conflicts < best_value) {  
        best_value = conflicts;  
        best_solution = new Solution(mgr);  
        idle_iter := 0;  
    }  
    else  
        idle_iter++;  
    iter++;  
}  
best_solution.restore(); // restore the variables assignment  
cout << coloring << endl;
```

◀ Go back



Adcock, S. (2005).

Genetic algorithms utility library.
Web Page.



Cahon, S., Melab, N., and Talbi, E. G. (2004).

ParadisEO: A framework for the reusable design of parallel and distributed metaheuristics.
Journal of Heuristics, 10(3):357–380.



Di Gaspero, L., Roli, A., and Schaerf, A. (2007).

Easyanalyzer: An object-oriented framework for the experimental analysis of stochastic local search algorithms.

In *Engineering Stochastic Local Search Algorithms. Designing, Implementing and Analyzing Effective Heuristics (SLS 2007)*, volume 4638 of *Lecture Notes in Computer Science*, pages 76–90.



Di Gaspero, L. and Schaerf, A. (2003).

EASYLOCAL++: An object-oriented framework for flexible design of local search algorithms.



Di Gaspero, L. and Schaerf, A. (2007).

Easysyn++: A tool for automatic synthesis of stochastic local search algorithms.

In *Engineering Stochastic Local Search Algorithms.*

Designing, Implementing and Analyzing Effective Heuristics (SLS 2007), volume 4638 of *Lecture Notes in Computer Science*, pages 177–181.



Dorne, R. and Voudouris, C. (2004).

HSF: the iOpt's framework to easily design metaheuristic methods, pages 237–256.

Kluwer Academic Publishers, Norwell, MA, USA.



Fink, A. and Voß, S. (2002).

HotFrame: A heuristic optimization framework.

In (Voß and Woodruff, 2002), pages 81–154.



Harder, R., Hill, R., and Moore, J. (2004).

A java universal vehicle router for routing unmanned vehicles.

International Transactions in Operations Research,
11:259–275.



Hoos, H. H. and Stützle, T. (2005).

Stochastic Local Search – Foundations and Applications.

Morgan Kaufmann Publishers, San Francisco, CA
(USA).



Laburthe, F. and Caseau, Y. (2002).

Salsa: A language for search algorithms.
Constraints, 7(3-4):255–288.



Lau, H. C., Wan, W. C., Halim, S., and Toh, K. Y. (2007).

A software framework for rapid hybridization of
meta-heuristics.

International Transactions in Operations Research,
14(2).



Michel, L. and Van Hentenryck, P. (2000).

Localizer.

Constraints, 5(1-2):43–84.

-  Shaw, P., De Backer, B., and Furnon, V. (2002).
Improved local search for cp toolkits.
Annals of Operations Research, 20(1–4):31–50.
-  Van Hentenryck, P. and Michel, L. (2005).
Constraint-based Local Search.
MIT Press.
-  Voß, S. and Woodruff, D. L., editors (2002).
Optimization Software Class Libraries.
OR/CS. Kluwer Academic Publishers, Dordrecht, the Netherlands.
-  Wall, M. B. (1996).
A Genetic Algorithm for Resource-Constrained Scheduling.
PhD thesis, MIT Mechanical Engineering Department.
-  Watson, J.-P. (2007).
The templated metaheuristic framework.
In *Proceedings of the 7th Metaheuristics International Conference (MIC 2007)*.